Project1-Blank

April 13, 2021

0.1 Introduction

Welcome to **CS188 - Data Science Fundamentals!** This course is designed to equip you with the tools and experiences necessary to start you off on a life-long exploration of datascience. We do not assume a prerequisite knowledge or experience in order to take the course.

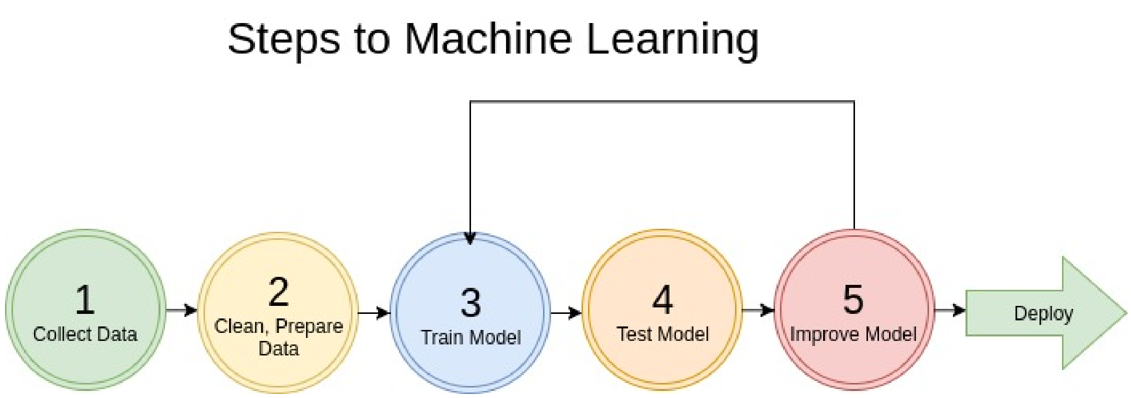
For this first project we will introduce you to the end-to-end process of doing a datascience project. Our goals for this project are to:

1. Familiarize you with the development environment for doing datascience
2. Get you comfortable with the python coding required to do datascience
3. Provide you with an sample end-to-end project to help you visualize the steps needed to complete a project on your own
4. Ask you to recreate a similar project on a separate dataset

In this project you will work through an example project end to end. Many of the concepts you will encounter will be unclear to you. That is OK! The course is designed to teach you these concepts in further detail. For now our focus is simply on having you replicate the code successfully and seeing a project through from start to finish.

Here are the main steps:

1. Get the data
2. Visualize the data for insights
3. Preprocess the data for your machine learning algorithm
4. Select a model and train
5. Does it meet the requirements? Fine tune the model



0.2 Working with Real Data

It is best to experiment with real-data as opposed to aritifical datasets.

There are many different open datasets depending on the type of problems you might be interested in!

Here are a few data repositories you could check out: - UCI Datasets - Kaggle Datasets - AWS Datasets

0.3 Submission Instructions

When you have completed this assignment please save the notebook as a PDF file and submit the assignment via Gradescope

1 Example Datascience Exercise

Below we will run through an California Housing example collected from the 1990's.

1.1 Setup

[2]:
```python
import sys
assert sys.version_info >= (3, 5) # python>=3.5
import sklearn
assert sklearn.__version__ >= "0.20" # sklearn >= 0.20

import numpy as np #numerical package in python
import os
%matplotlib inline
import matplotlib.pyplot as plt #plotting package

# to make this notebook's output identical at every run
np.random.seed(42)

#matplotlib magic for inline figures
%matplotlib inline
import matplotlib # plotting library
import matplotlib.pyplot as plt

# Where to save the figures
ROOT_DIR = "."
IMAGES_PATH = os.path.join(ROOT_DIR, "images")
os.makedirs(IMAGES_PATH, exist_ok=True)

def save_fig(fig_name, tight_layout=True, fig_extension="png", resolution=300):
    '''
        plt.savefig wrapper. refer to
        https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.savefig.html
```

```
        Args:
            fig_name (str): name of the figrue
            tight_layout (bool): adjust subplot to fit in the figure area
            fig_extension (str): file format to save the figure in
            resolution (int): figure resolution
    '''
    path = os.path.join(IMAGES_PATH, fig_name + "." + fig_extension)
    print("Saving figure", fig_name)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

[3]:
```
import os
import tarfile
import urllib
DATASET_PATH = os.path.join("datasets", "housing")
```

1.2 Step 1. Getting the data

1.2.1 Intro to Data Exploration Using Pandas

In this section we will load the dataset, and visualize different features using different types of plots.

Packages we will use: - **Pandas:** is a fast, flexibile and expressive data structure widely used for tabular and multidimensional datasets. - **Matplotlib**: is a 2d python plotting library which you can use to create quality figures (you can plot almost anything if you're willing to code it out!) - other plotting libraries:seaborn, ggplot2

[4]:
```
import pandas as pd

def load_housing_data(housing_path):
    '''
        loads housing.csv dataset stored

        Args:
            housing_path (str): path to folder containing housing datased

        Returns:
            pd.DataFrame
    '''
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)
```

[5]:
```
pd.DataFrame
```

[5]:
```
pandas.core.frame.DataFrame
```

[6]:
```python
housing = load_housing_data(DATASET_PATH) # we load the pandas dataframe
housing.head() # show the first few elements of the dataframe
               # typically this is the first thing you do
               # to see how the dataframe looks like
```

[6]:
```
   longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
0    -122.23     37.88                41.0        880.0           129.0
1    -122.22     37.86                21.0       7099.0          1106.0
2    -122.24     37.85                52.0       1467.0           190.0
3    -122.25     37.85                52.0       1274.0           235.0
4    -122.25     37.85                52.0       1627.0           280.0

   population  households  median_income  median_house_value ocean_proximity
0       322.0       126.0         8.3252            452600.0        NEAR BAY
1      2401.0      1138.0         8.3014            358500.0        NEAR BAY
2       496.0       177.0         7.2574            352100.0        NEAR BAY
3       558.0       219.0         5.6431            341300.0        NEAR BAY
4       565.0       259.0         3.8462            342200.0        NEAR BAY
```

A dataset may have different types of features - real valued - Discrete (integers) - categorical (strings)

The two categorical features are essentialy the same as you can always map a categorical string/character to an integer.

In the dataset example, all our features are real valued floats, except ocean proximity which is categorical.

[7]:
```python
# to see a concise summary of data types, null values, and counts
# use the info() method on the dataframe
housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  float64
 3   total_rooms         20640 non-null  float64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  float64
 6   households          20640 non-null  float64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  float64
 9   ocean_proximity     20640 non-null  object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

[8]:

```python
# you can access individual columns similarly
# to accessing elements in a python dict
housing["ocean_proximity"].head() # added head() to avoid printing many columns.
↪.
```

[8]:

```
0    NEAR BAY
1    NEAR BAY
2    NEAR BAY
3    NEAR BAY
4    NEAR BAY
Name: ocean_proximity, dtype: object
```

[9]:

```python
# to access a particular row we can use iloc
housing.iloc[1]
```

[9]:

```
longitude              -122.22
latitude                 37.86
housing_median_age          21
total_rooms               7099
total_bedrooms            1106
population                2401
households                1138
median_income           8.3014
median_house_value      358500
ocean_proximity       NEAR BAY
Name: 1, dtype: object
```

[10]:

```python
# one other function that might be useful is
# value_counts(), which counts the number of occurences
# for categorical features
housing["ocean_proximity"].value_counts()
```

[10]:

```
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

[11]:

```python
# The describe function compiles your typical statistics for each
# column
housing.describe()
```

[11]:

```
          longitude      latitude  housing_median_age   total_rooms  \
count  20640.000000  20640.000000        20640.000000  20640.000000
mean    -119.569704     35.631861           28.639486   2635.763081
std        2.003532      2.135952           12.585558   2181.615252
```

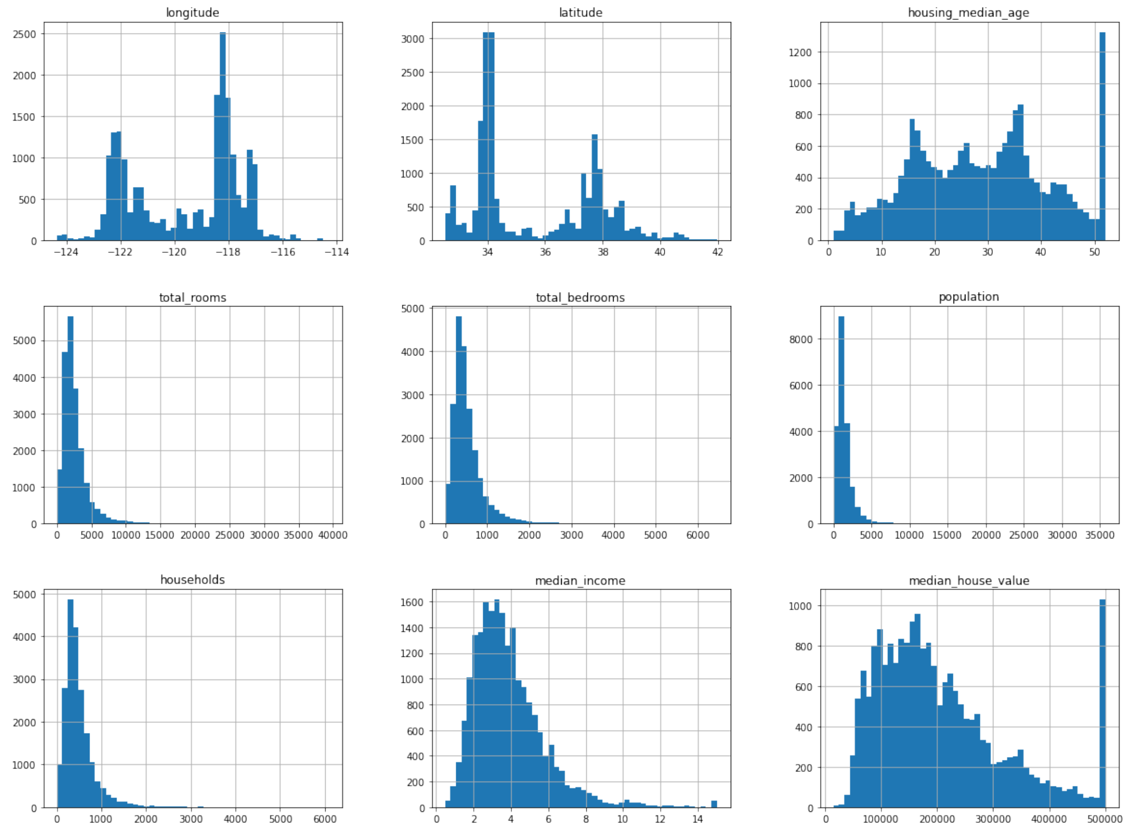
```
min    -124.350000    32.540000          1.000000      2.000000
25%    -121.800000    33.930000         18.000000   1447.750000
50%    -118.490000    34.260000         29.000000   2127.000000
75%    -118.010000    37.710000         37.000000   3148.000000
max    -114.310000    41.950000         52.000000  39320.000000

       total_bedrooms    population    households  median_income  \
count    20433.000000  20640.000000  20640.000000   20640.000000
mean       537.870553   1425.476744    499.539680       3.870671
std        421.385070   1132.462122    382.329753       1.899822
min          1.000000      3.000000      1.000000       0.499900
25%        296.000000    787.000000    280.000000       2.563400
50%        435.000000   1166.000000    409.000000       3.534800
75%        647.000000   1725.000000    605.000000       4.743250
max       6445.000000  35682.000000   6082.000000      15.000100

       median_house_value
count        20640.000000
mean        206855.816909
std         115395.615874
min          14999.000000
25%         119600.000000
50%         179700.000000
75%         264725.000000
max         500001.000000
```

If you want to learn about different ways of accessing elements or other functions it's useful to check out the getting started section here

1.3 Step 2. Visualizing the data

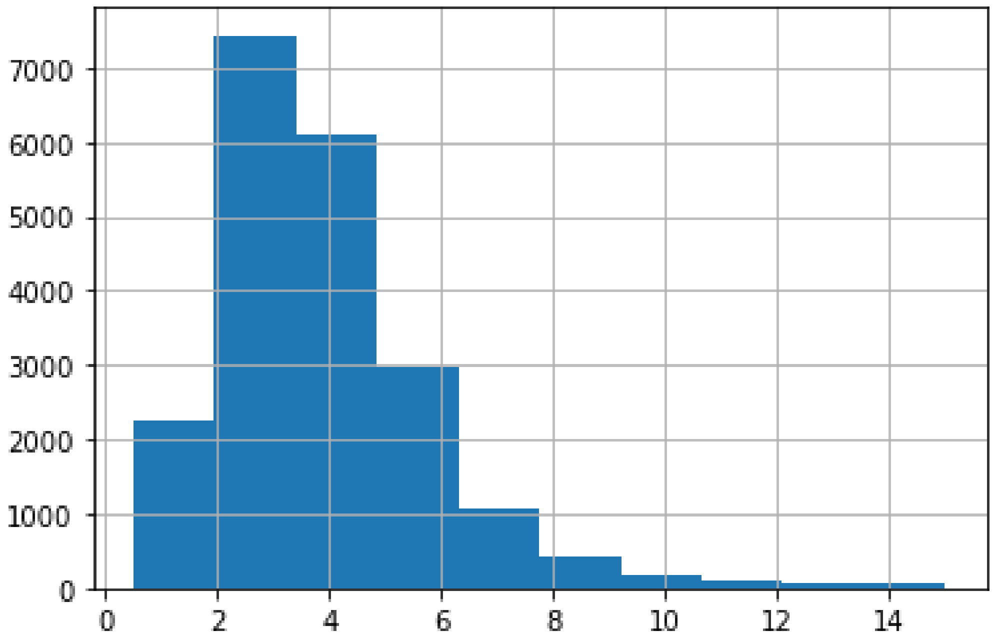
1.3.1 Let's start visualizing the dataset

[12]:
```python
# We can draw a histogram for each of the dataframes features
# using the hist function
housing.hist(bins=50, figsize=(20,15))
# save_fig("attribute_histogram_plots")
plt.show() # pandas internally uses matplotlib, and to display all the figures
           # the show() function must be called
```



[13]:

```
# if you want to have a histogram on an individual feature:
housing["median_income"].hist() # default is 10 bins
plt.show()
```



We can convert a floating point feature to a categorical feature by binning or by defining a set of intervals.

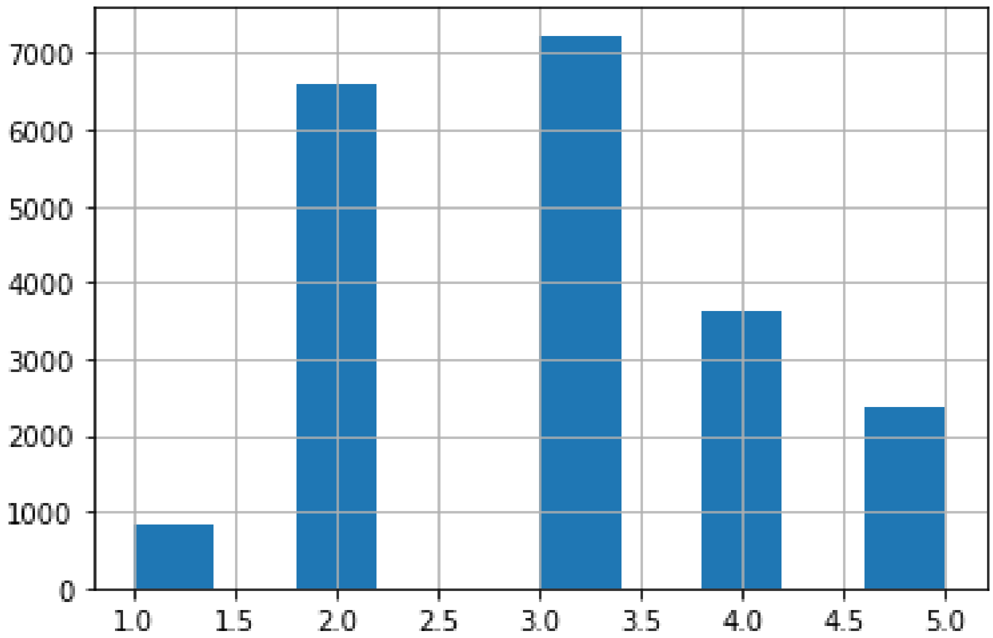
For example, to bin the households based on median_income we can use the pd.cut function

[14]:
```
# assign each bin a categorical value [1, 2, 3, 4, 5] in this case.
housing["income_cat"] = pd.cut(housing["median_income"],
                               bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                               labels=[1, 2, 3, 4, 5])

housing["income_cat"].value_counts()
```

[14]:
```
3    7236
2    6581
4    3639
5    2362
1     822
Name: income_cat, dtype: int64
```

[15]:
```
housing["income_cat"].hist()
```

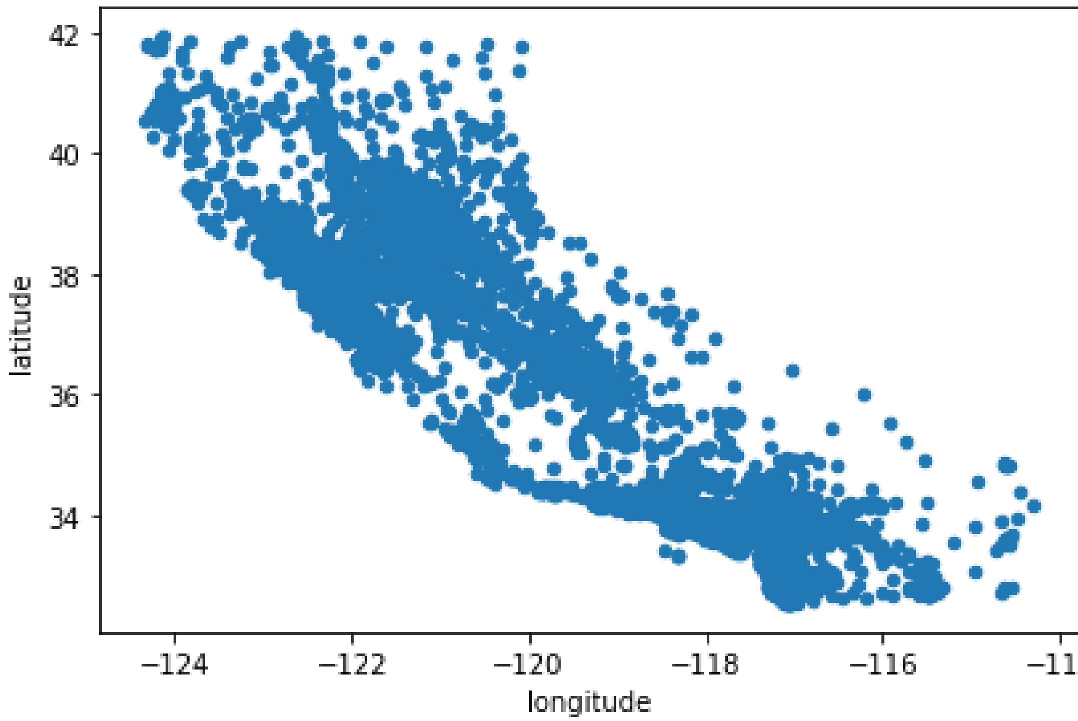
[15]:
```
<AxesSubplot:>
```



Next let's visualize the household incomes based on latitude & longitude coordinates

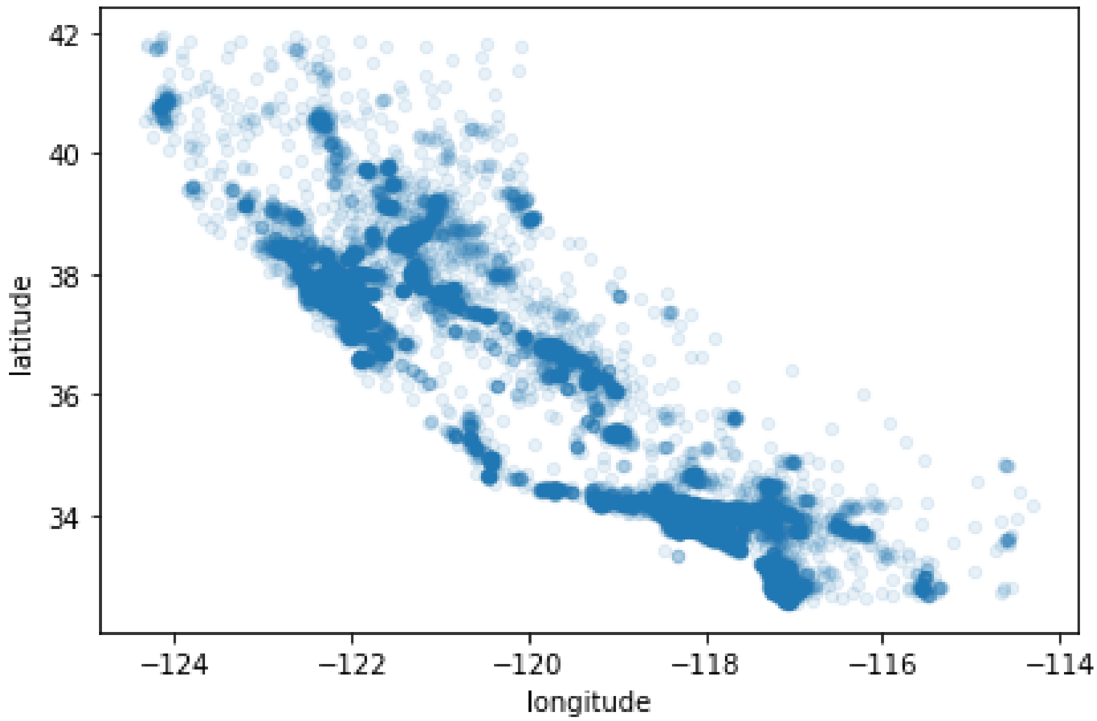
[16]:
```python
## here's a not so interestting way of plotting it
housing.plot(kind="scatter", x="longitude", y="latitude")
save_fig("bad_visualization_plot")
```

```
Saving figure bad_visualization_plot
```




```
[17]: # we can make it look a bit nicer by using the alpha parameter,
      # it simply plots less dense areas lighter.
      housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
      save_fig("better_visualization_plot")
```

```
Saving figure better_visualization_plot
```

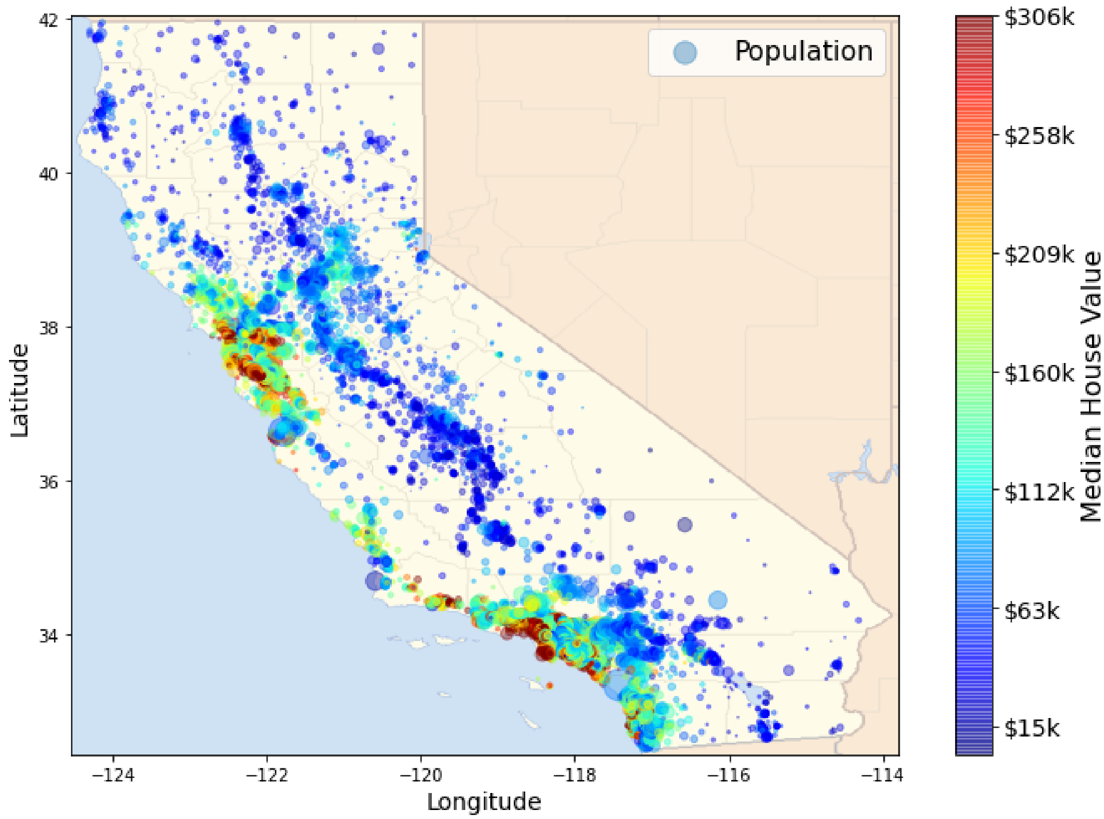


[18]:
```
# A more interesting plot is to color code (heatmap) the dots
# based on income. The code below achieves this

# load an image of california
images_path = os.path.join('./', "images")
os.makedirs(images_path, exist_ok=True)
filename = "california.png"

import matplotlib.image as mpimg
california_img=mpimg.imread(os.path.join(images_path, filename))
ax = housing.plot(kind="scatter", x="longitude", y="latitude", figsize=(10,7),
                       s=housing['population']/100, label="Population",
                       c="median_house_value", cmap=plt.get_cmap("jet"),
                       colorbar=False, alpha=0.4,
                      )
# note above how if we remove colorbar=False above, a duplicate colorbar will␣
 ↪appear

# overlay the califronia map on the plotted scatter plot
# note: plt.imshow still refers to the most recent figure
# that hasn't been plotted yet.
plt.imshow(california_img, extent=[-124.55, -113.80, 32.45, 42.05], alpha=0.5,
           cmap=plt.get_cmap("jet"))
```

```python
plt.ylabel("Latitude", fontsize=14)
plt.xlabel("Longitude", fontsize=14)

# setting up heatmap colors based on median_house_value feature
prices = housing["median_house_value"]
tick_values = np.linspace(prices.min(), prices.max(), 11)
print(tick_values)
cb = plt.colorbar() # add a colorbar to plot
# %d is a formatter for integers; k is to represent "thousand" in the scale
cb.ax.set_yticklabels(["$%dk"%(round(v/1000)) for v in tick_values],␣
 ↪fontsize=14)
cb.set_label('Median House Value', fontsize=16)

# Why are there only 7 ticks in the colorbar below but our linspace function␣
 ↪above divided it into 11 evenly-spaced tick values???

plt.legend(fontsize=16)
save_fig("california_housing_prices_plot")
plt.show()
```

```
[ 14999.   63499.2 111999.4 160499.6 208999.8 257500.  306000.2 354500.4
 403000.6 451500.8 500001. ]
Saving figure california_housing_prices_plot
```

```
<ipython-input-18-f361eec34593>:32: UserWarning: FixedFormatter should only be
used together with FixedLocator
  cb.ax.set_yticklabels(["$%dk"%(round(v/1000)) for v in tick_values],
fontsize=14)
```



Not suprisingly, we can see that the most expensive houses are concentrated around the San Francisco/Los Angeles areas.

Up until now we have only visualized feature histograms and basic statistics.

When developing machine learning models the predictiveness of a feature for a particular target of intrest is what's important.

It may be that only a few features are useful for the target at hand, or features may need to be augmented by applying certain transformations.

None the less we can explore this using correlation matrices. If you need to brush up on correlation take a look here.

[19]:
```
corr_matrix = housing.corr() # compute the correlation matrix
```

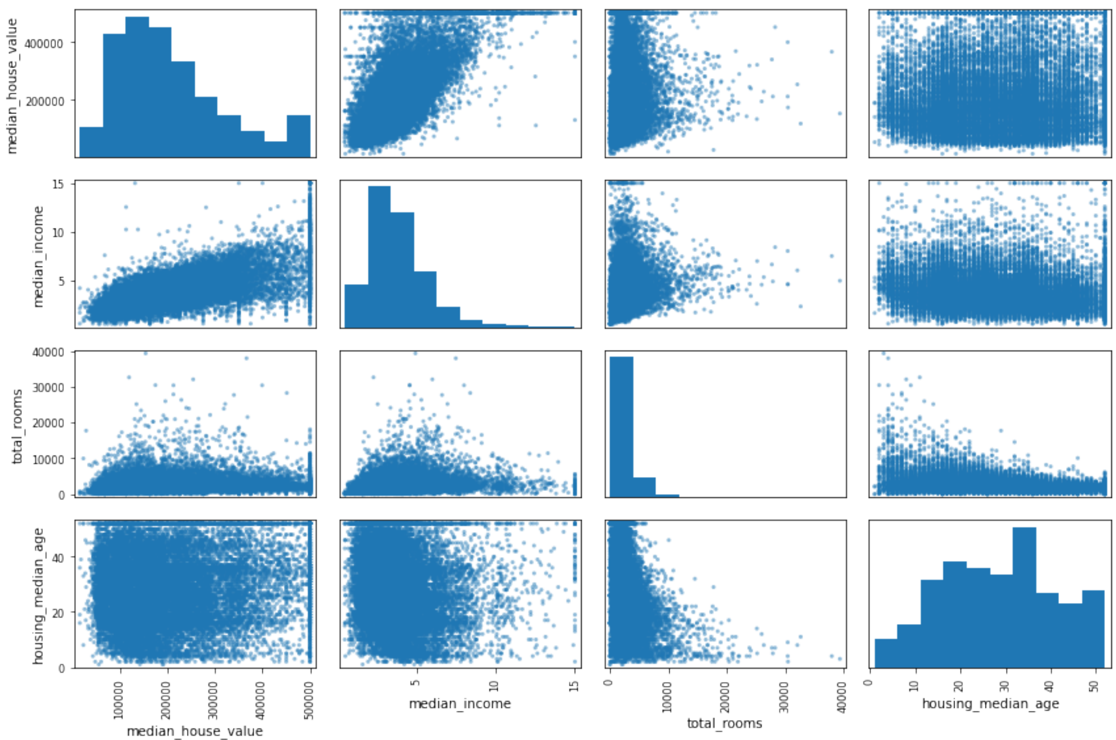
[20]:
```
# for example if the target is "median_house_value", most correlated features␣
↪can be sorted
# which happens to be "median_income". This also intuitively makes sense.
corr_matrix["median_house_value"].sort_values(ascending=False)
```

[20]:
```
median_house_value    1.000000
median_income         0.688075
```

```
total_rooms           0.134153
housing_median_age    0.105623
households            0.065843
total_bedrooms        0.049686
population           -0.024650
longitude            -0.045967
latitude             -0.144160
Name: median_house_value, dtype: float64
```

[21]:
```python
# the correlation matrix for different attributes/features can also be plotted
# some features may show a positive correlation/negative correlation or
# it may turn out to be completely random!
from pandas.plotting import scatter_matrix
attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
save_fig("scatter_matrix_plot")
```

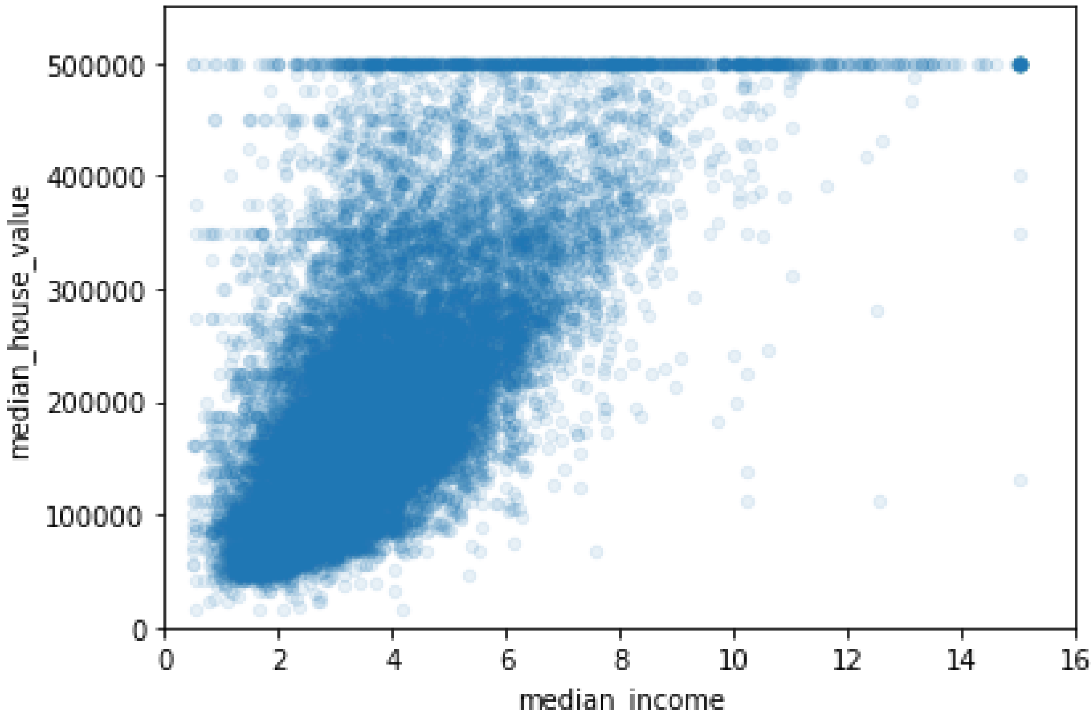
```
Saving figure scatter_matrix_plot
```



[22]:
```python
# median income vs median house vlue plot plot 2 in the first row of top figure
housing.plot(kind="scatter", x="median_income", y="median_house_value",
             alpha=0.1)
```

```
plt.axis([0, 16, 0, 550000])
save_fig("income_vs_house_value_scatterplot")
```

```
Saving figure income_vs_house_value_scatterplot
```



1.3.2 Augmenting Features

New features can be created by combining different columns from our data set.

- rooms_per_household = total_rooms / households
- bedrooms_per_room = total_bedrooms / total_rooms
- etc.

[23]:
```
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"]=housing["population"]/housing["households"]
```

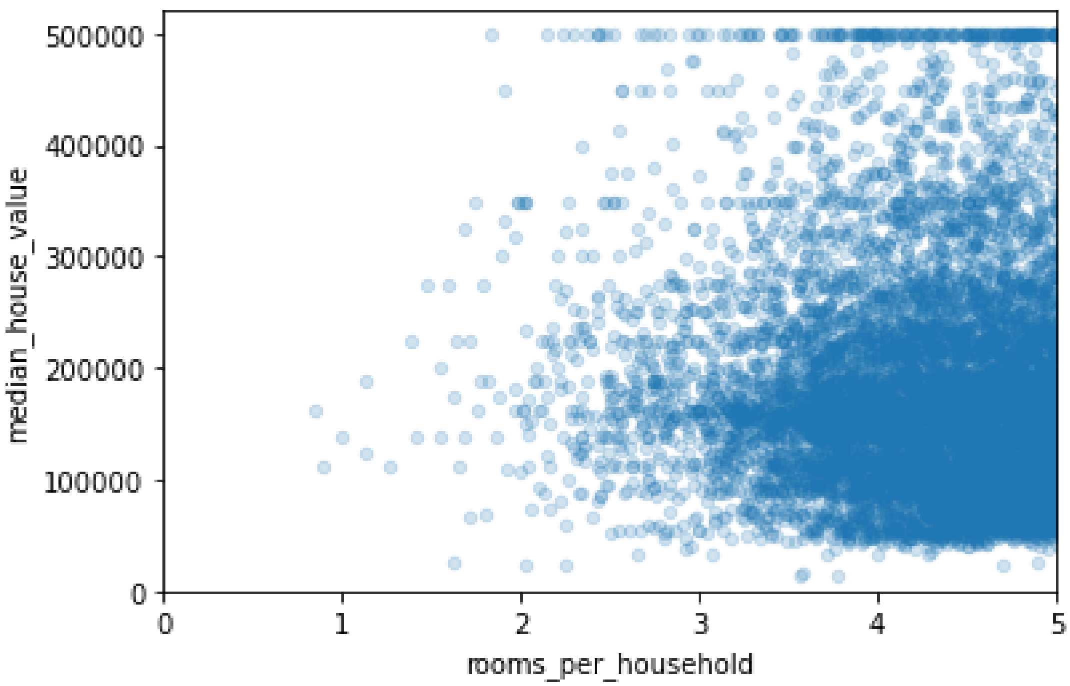
[24]:
```
# obtain new correlations
corr_matrix = housing.corr()
corr_matrix["median_house_value"].sort_values(ascending=False)
```

[24]:
```
median_house_value          1.000000
median_income               0.688075
rooms_per_household         0.151948
```

```
total_rooms                 0.134153
housing_median_age          0.105623
households                  0.065843
total_bedrooms              0.049686
population_per_household   -0.023737
population                 -0.024650
longitude                  -0.045967
latitude                   -0.144160
bedrooms_per_room          -0.255880
Name: median_house_value, dtype: float64
```

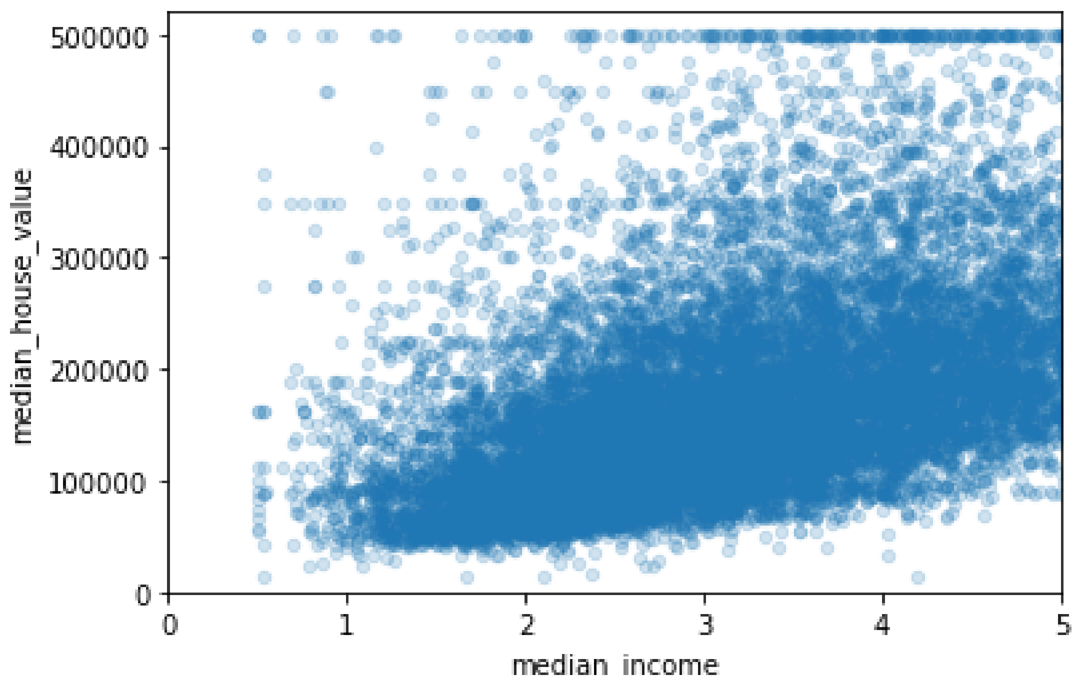
[25]:

```
housing.plot(kind="scatter", x="rooms_per_household", y="median_house_value",
             alpha=0.2)
plt.axis([0, 5, 0, 520000])
plt.show()
```



[26]:

```
housing.plot(kind="scatter", x="median_income", y="median_house_value",
             alpha=0.2)
plt.axis([0, 5, 0, 520000])
plt.show()
```



[27]:
```
housing.describe()
```

[27]:
```
          longitude      latitude  housing_median_age   total_rooms  \
count  20640.000000  20640.000000        20640.000000  20640.000000
mean    -119.569704     35.631861           28.639486   2635.763081
std        2.003532      2.135952           12.585558   2181.615252
min     -124.350000     32.540000            1.000000      2.000000
25%     -121.800000     33.930000           18.000000   1447.750000
50%     -118.490000     34.260000           29.000000   2127.000000
75%     -118.010000     37.710000           37.000000   3148.000000
max     -114.310000     41.950000           52.000000  39320.000000

       total_bedrooms    population    households  median_income  \
count    20433.000000  20640.000000  20640.000000   20640.000000
mean       537.870553   1425.476744    499.539680       3.870671
std        421.385070   1132.462122    382.329753       1.899822
min          1.000000      3.000000      1.000000       0.499900
25%        296.000000    787.000000    280.000000       2.563400
50%        435.000000   1166.000000    409.000000       3.534800
75%        647.000000   1725.000000    605.000000       4.743250
max       6445.000000  35682.000000   6082.000000      15.000100

       median_house_value  rooms_per_household  bedrooms_per_room  \
count        20640.000000         20640.000000       20433.000000
```

```
mean        206855.816909          5.429000          0.213039
std         115395.615874          2.474173          0.057983
min          14999.000000          0.846154          0.100000
25%         119600.000000          4.440716          0.175427
50%         179700.000000          5.229129          0.203162
75%         264725.000000          6.052381          0.239821
max         500001.000000        141.909091          1.000000

       population_per_household
count              20640.000000
mean                   3.070655
std                   10.386050
min                    0.692308
25%                    2.429741
50%                    2.818116
75%                    3.282261
max                 1243.333333
```

1.4 Step 3. Preprocess the data for your machine learning algorithm

Once we've visualized the data, and have a certain understanding of how the data looks like. It's time to clean!

Most of your time will be spent on this step, although the datasets used in this project are relatively nice and clean… in the real world it could get real dirty.

After having cleaned your dataset you're aiming for: - train set - test set

In some cases you might also have a validation set as well for tuning hyperparameters (don't worry if you're not familiar with this term yet..)

In supervised learning setting your train set and test set should contain (**feature**, **target**) tuples. - **feature**: is the input to your model - **target**: is the ground truth label - when target is categorical the task is a classification task - when target is floating point the task is a regression task - I don't really understand this. Why is it a floating point, and what qualifies as a regression task for supervised learning???

We will make use of **scikit-learn** python package for preprocessing.

Scikit learn is pretty well documented and if you get confused at any point simply look up the function/object!

1.4.1 Dealing With Incomplete Data

[28]:
```
# have you noticed when looking at the dataframe summary certain rows
# contained null values? we can't just leave them as nulls and expect our
# model to handle them for us so we'll have to devise a method for dealing with␣
↪them...
# I'm a little confused by what pd.any() does???
sample_incomplete_rows = housing[housing.isnull().any(axis=1)].head()
```

```
sample_incomplete_rows
```

[28]:
```
     longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
290    -122.16     37.77                47.0       1256.0             NaN
341    -122.17     37.75                38.0        992.0             NaN
538    -122.28     37.78                29.0       5154.0             NaN
563    -122.24     37.75                45.0        891.0             NaN
696    -122.10     37.69                41.0        746.0             NaN

     population  households  median_income  median_house_value  \
290       570.0       218.0         4.3750            161900.0
341       732.0       259.0         1.6196             85100.0
538      3741.0      1273.0         2.5762            173400.0
563       384.0       146.0         4.9489            247100.0
696       387.0       161.0         3.9063            178400.0

    ocean_proximity income_cat  rooms_per_household  bedrooms_per_room  \
290        NEAR BAY          3             5.761468                NaN
341        NEAR BAY          2             3.830116                NaN
538        NEAR BAY          2             4.048704                NaN
563        NEAR BAY          4             6.102740                NaN
696        NEAR BAY          3             4.633540                NaN

     population_per_household
290                  2.614679
341                  2.826255
538                  2.938727
563                  2.630137
696                  2.403727
```

[29]:
```
sample_incomplete_rows.dropna(subset=["total_bedrooms"])     # option 1: simply␣
↪drop rows that have null values
```

[29]:
```
Empty DataFrame
Columns: [longitude, latitude, housing_median_age, total_rooms, total_bedrooms,
population, households, median_income, median_house_value, ocean_proximity,
income_cat, rooms_per_household, bedrooms_per_room, population_per_household]
Index: []
```

[30]:
```
sample_incomplete_rows.drop("total_bedrooms", axis=1)       # option 2: drop␣
↪the complete feature
```

[30]:
```
     longitude  latitude  housing_median_age  total_rooms  population  \
290    -122.16     37.77                47.0       1256.0       570.0
341    -122.17     37.75                38.0        992.0       732.0
538    -122.28     37.78                29.0       5154.0      3741.0
563    -122.24     37.75                45.0        891.0       384.0
```

```
696    -122.10     37.69                41.0        746.0       387.0

     households  median_income  median_house_value ocean_proximity income_cat  \
290       218.0         4.3750            161900.0        NEAR BAY          3
341       259.0         1.6196             85100.0        NEAR BAY          2
538      1273.0         2.5762            173400.0        NEAR BAY          2
563       146.0         4.9489            247100.0        NEAR BAY          4
696       161.0         3.9063            178400.0        NEAR BAY          3

     rooms_per_household  bedrooms_per_room  population_per_household
290             5.761468                NaN                  2.614679
341             3.830116                NaN                  2.826255
538             4.048704                NaN                  2.938727
563             6.102740                NaN                  2.630137
696             4.633540                NaN                  2.403727
```

[31]:

```
median = housing["total_bedrooms"].median()
sample_incomplete_rows["total_bedrooms"].fillna(median, inplace=True) # option␣
↪3: replace na values with median values
sample_incomplete_rows
```

[31]:

```
     longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
290    -122.16     37.77                47.0       1256.0           435.0
341    -122.17     37.75                38.0        992.0           435.0
538    -122.28     37.78                29.0       5154.0           435.0
563    -122.24     37.75                45.0        891.0           435.0
696    -122.10     37.69                41.0        746.0           435.0

     population  households  median_income  median_house_value  \
290       570.0       218.0         4.3750            161900.0
341       732.0       259.0         1.6196             85100.0
538      3741.0      1273.0         2.5762            173400.0
563       384.0       146.0         4.9489            247100.0
696       387.0       161.0         3.9063            178400.0

    ocean_proximity income_cat  rooms_per_household  bedrooms_per_room  \
290        NEAR BAY          3             5.761468                NaN
341        NEAR BAY          2             3.830116                NaN
538        NEAR BAY          2             4.048704                NaN
563        NEAR BAY          4             6.102740                NaN
696        NEAR BAY          3             4.633540                NaN

     population_per_household
290                  2.614679
341                  2.826255
538                  2.938727
563                  2.630137
```

```
696                     2.403727
```

Could you think of another plausible imputation for this dataset? (Not graded)

1.4.2 Prepare Data

Recall we are trying to predict the median house value, our features will contain longitude, latitude, housing_median_age… and our target will be median_house_value

[32]:
```
housing_features = housing.drop("median_house_value", axis=1) # drop labels for␣
↪training set features
                                                        # the input to the model␣
↪should not contain the true label
housing_labels = housing["median_house_value"].copy()
```

[33]:
```
housing_features.head()
```

[33]:
```
   longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
0    -122.23     37.88                41.0        880.0           129.0
1    -122.22     37.86                21.0       7099.0          1106.0
2    -122.24     37.85                52.0       1467.0           190.0
3    -122.25     37.85                52.0       1274.0           235.0
4    -122.25     37.85                52.0       1627.0           280.0

   population  households  median_income ocean_proximity income_cat  \
0       322.0       126.0         8.3252        NEAR BAY          5
1      2401.0      1138.0         8.3014        NEAR BAY          5
2       496.0       177.0         7.2574        NEAR BAY          5
3       558.0       219.0         5.6431        NEAR BAY          4
4       565.0       259.0         3.8462        NEAR BAY          3

   rooms_per_household  bedrooms_per_room  population_per_household
0             6.984127           0.146591                  2.555556
1             6.238137           0.155797                  2.109842
2             8.288136           0.129516                  2.802260
3             5.817352           0.184458                  2.547945
4             6.281853           0.172096                  2.181467
```

[34]:
```
# This cell implements the complete pipeline for preparing the data
# using sklearns TransformerMixins
# Earlier we mentioned different types of features: categorical, and floats.
# In the case of floats we might want to convert them to categories.
# On the other hand categories in which are not already represented as integers␣
↪must be mapped to integers before
# feeding to the model.
```

```
# Additionally, categorical values could either be represented as one-hot␣
↪vectors or simple as normalized/unnormalized integers.
# Here we encode them using one hot vectors.

# DO NOT WORRY IF YOU DO NOT UNDERSTAND ALL THE STEPS OF THIS PIPELINE.␣
↪CONCEPTS LIKE NORMALIZATION,
# ONE-HOT ENCODING ETC. WILL ALL BE COVERED IN DISCUSSION

from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder

from sklearn.base import BaseEstimator, TransformerMixin


imputer = SimpleImputer(strategy="median") # use median imputation for missing␣
↪values
housing_num = housing_features.drop("ocean_proximity", axis=1) # remove the␣
↪categorical feature
# column index
rooms_idx, bedrooms_idx, population_idx, households_idx = 3, 4, 5, 6

#
class AugmentFeatures(BaseEstimator, TransformerMixin):
    '''
    implements the previous features we had defined
    housing["rooms_per_household"] = housing["total_rooms"]/
↪housing["households"]
    housing["bedrooms_per_room"] = housing["total_bedrooms"]/
↪housing["total_rooms"]
    housing["population_per_household"]=housing["population"]/
↪housing["households"]
    '''
    def __init__(self, add_bedrooms_per_room = True):
        self.add_bedrooms_per_room = add_bedrooms_per_room

    def fit(self, X, y=None):
        return self  # nothing else to do

    def transform(self, X):
        rooms_per_household = X[:, rooms_idx] / X[:, households_idx]
        population_per_household = X[:, population_idx] / X[:, households_idx]
        if self.add_bedrooms_per_room:
```

```
            bedrooms_per_room = X[:, bedrooms_idx] / X[:, rooms_idx]
            return np.c_[X, rooms_per_household, population_per_household,
                         bedrooms_per_room]
        else:
            return np.c_[X, rooms_per_household, population_per_household]

attr_adder = AugmentFeatures(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values) # generate new␣
 ↪features

# this will be are numirical pipeline
# 1. impute, 2. augment the feature set 3. normalize using StandardScaler()
num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', AugmentFeatures()),
        ('std_scaler', StandardScaler()),
    ])

housing_num_tr = num_pipeline.fit_transform(housing_num)

numerical_features = list(housing_num)
categorical_features = ["ocean_proximity"]

full_pipeline = ColumnTransformer([
        ("num", num_pipeline, numerical_features),
        ("cat", OneHotEncoder(), categorical_features),
    ])

housing_prepared = full_pipeline.fit_transform(housing_features)
```

1.4.3 Splitting our dataset

First we need to carve out our dataset into a training and testing cohort. To do this we'll use train_test_split, a very elementary tool that arbitrarily splits the data into training and testing cohorts.

[35]:
```
from sklearn.model_selection import train_test_split
data_target = housing['median_house_value']
train, test, target, target_test = train_test_split(housing_prepared,␣
 ↪data_target, test_size=0.3, random_state=0)
```

1.4.4 Select a model and train

Once we have prepared the dataset it's time to choose a model.

As our task is to predict the median_house_value (a floating value), regression is well suited for this.

[36]:
```python
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(train, target)

# let's try the full preprocessing pipeline on a few training instances
data = test
labels = target_test

print("Predictions:", lin_reg.predict(data)[:5])
print("Actual labels:", list(labels)[:5])
```

```
Predictions: [207828.06448011 281099.80175494 176021.36890539  93643.46744928
 304674.47047758]
Actual labels: [136900.0, 241300.0, 200700.0, 72500.0, 460000.0]
```

[37]:
```python
from sklearn.metrics import mean_squared_error

preds = lin_reg.predict(test)
mse = mean_squared_error(target_test, preds)
rmse = np.sqrt(mse)
rmse
```

[37]: 67879.86844243006

2 TODO: Applying the end-end ML steps to a different dataset.

We will apply what we've learnt to another dataset (airbnb dataset). We will predict airbnb price based on other features.

3 [35 pts] Visualizing Data

3.0.1 [5 pts] Load the data + statistics

- load the dataset
- display the first few rows of the data

[110]:
```python
def load_airbnb_data(airbnb_path):
    '''
        loads AB_NYC_2019.csv dataset stored

        Args:
            airbnb_path (str): path to folder containing airbnb dataset

        Returns:
            pd.DataFrame
    '''
```

```
    csv_path = os.path.join(airbnb_path, "AB_NYC_2019.csv")
    return pd.read_csv(csv_path)

DATASET_PATH = os.path.join("datasets", "airbnb")
airbnb = load_airbnb_data(DATASET_PATH)
airbnb.head()
```

[110]:
```
     id                                              name  host_id  \
0  2539                Clean & quiet apt home by the park     2787
1  2595                             Skylit Midtown Castle     2845
2  3647               THE VILLAGE OF HARLEM…NEW YORK !      4632
3  3831                   Cozy Entire Floor of Brownstone     4869
4  5022  Entire Apt: Spacious Studio/Loft by central park     7192

     host_name neighbourhood_group neighbourhood  latitude  longitude  \
0         John            Brooklyn    Kensington  40.64749  -73.97237
1     Jennifer           Manhattan       Midtown  40.75362  -73.98377
2    Elisabeth           Manhattan        Harlem  40.80902  -73.94190
3  LisaRoxanne            Brooklyn  Clinton Hill  40.68514  -73.95976
4        Laura           Manhattan   East Harlem  40.79851  -73.94399

         room_type  price  minimum_nights  number_of_reviews last_review  \
0     Private room    149               1                  9  2018-10-19
1  Entire home/apt    225               1                 45  2019-05-21
2     Private room    150               3                  0         NaN
3  Entire home/apt     89               1                270  2019-07-05
4  Entire home/apt     80              10                  9  2018-11-19

   reviews_per_month  calculated_host_listings_count  availability_365
0               0.21                               6               365
1               0.38                               2               355
2                NaN                               1               365
3               4.64                               1               194
4               0.10                               1                 0
```

- pull up info on the data type for each of the data fields. Will any of these be problemmatic feeding into your model (you may need to do a little research on this)? Discuss:

[111]:
```
airbnb.info()
# As we can see below, a few fields like name, host_name, last_review, and␣
↪reviews_per_month have some null values that we have to deal with.
# Some of the fields like neighborhood and room_type are categorical.
# What is the difference between neighborhood_group and neighborhood???
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48895 entries, 0 to 48894
Data columns (total 16 columns):
 #   Column                          Non-Null Count  Dtype
```

```
---  ------                          --------------  -----
 0   id                              48895 non-null  int64
 1   name                            48879 non-null  object
 2   host_id                         48895 non-null  int64
 3   host_name                       48874 non-null  object
 4   neighbourhood_group             48895 non-null  object
 5   neighbourhood                   48895 non-null  object
 6   latitude                        48895 non-null  float64
 7   longitude                       48895 non-null  float64
 8   room_type                       48895 non-null  object
 9   price                           48895 non-null  int64
 10  minimum_nights                  48895 non-null  int64
 11  number_of_reviews               48895 non-null  int64
 12  last_review                     38843 non-null  object
 13  reviews_per_month               38843 non-null  float64
 14  calculated_host_listings_count  48895 non-null  int64
 15  availability_365                48895 non-null  int64
dtypes: float64(3), int64(7), object(6)
memory usage: 6.0+ MB
```

[Response here]

- drop the following columns: name, host_id, host_name, and last_review
- display a summary of the statistics of the loaded data

[112]:
```python
airbnb.drop("name", axis=1, inplace=True)
airbnb.drop("host_id", axis=1, inplace=True)
airbnb.drop("host_name", axis=1, inplace=True)
airbnb.drop("last_review", axis=1, inplace=True)
airbnb.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48895 entries, 0 to 48894
Data columns (total 12 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   id                              48895 non-null  int64
 1   neighbourhood_group             48895 non-null  object
 2   neighbourhood                   48895 non-null  object
 3   latitude                        48895 non-null  float64
 4   longitude                       48895 non-null  float64
 5   room_type                       48895 non-null  object
 6   price                           48895 non-null  int64
 7   minimum_nights                  48895 non-null  int64
 8   number_of_reviews               48895 non-null  int64
 9   reviews_per_month               38843 non-null  float64
 10  calculated_host_listings_count  48895 non-null  int64
 11  availability_365                48895 non-null  int64
dtypes: float64(3), int64(6), object(3)
```

```
memory usage: 4.5+ MB
```

[113]:
```
airbnb.describe()
```

[113]:
```
                 id      latitude     longitude         price  minimum_nights  \
count  4.889500e+04  48895.000000  48895.000000  48895.000000    48895.000000
mean   1.901714e+07     40.728949    -73.952170    152.720687        7.029962
std    1.098311e+07      0.054530      0.046157    240.154170       20.510550
min    2.539000e+03     40.499790    -74.244420      0.000000        1.000000
25%    9.471945e+06     40.690100    -73.983070     69.000000        1.000000
50%    1.967728e+07     40.723070    -73.955680    106.000000        3.000000
75%    2.915218e+07     40.763115    -73.936275    175.000000        5.000000
max    3.648724e+07     40.913060    -73.712990  10000.000000     1250.000000

       number_of_reviews  reviews_per_month  calculated_host_listings_count  \
count       48895.000000       38843.000000                    48895.000000
mean           23.274466           1.373221                        7.143982
std            44.550582           1.680442                       32.952519
min             0.000000           0.010000                        1.000000
25%             1.000000           0.190000                        1.000000
50%             5.000000           0.720000                        1.000000
75%            24.000000           2.020000                        2.000000
max           629.000000          58.500000                      327.000000

       availability_365
count      48895.000000
mean         112.781327
std          131.622289
min            0.000000
25%            0.000000
50%           45.000000
75%          227.000000
max          365.000000
```

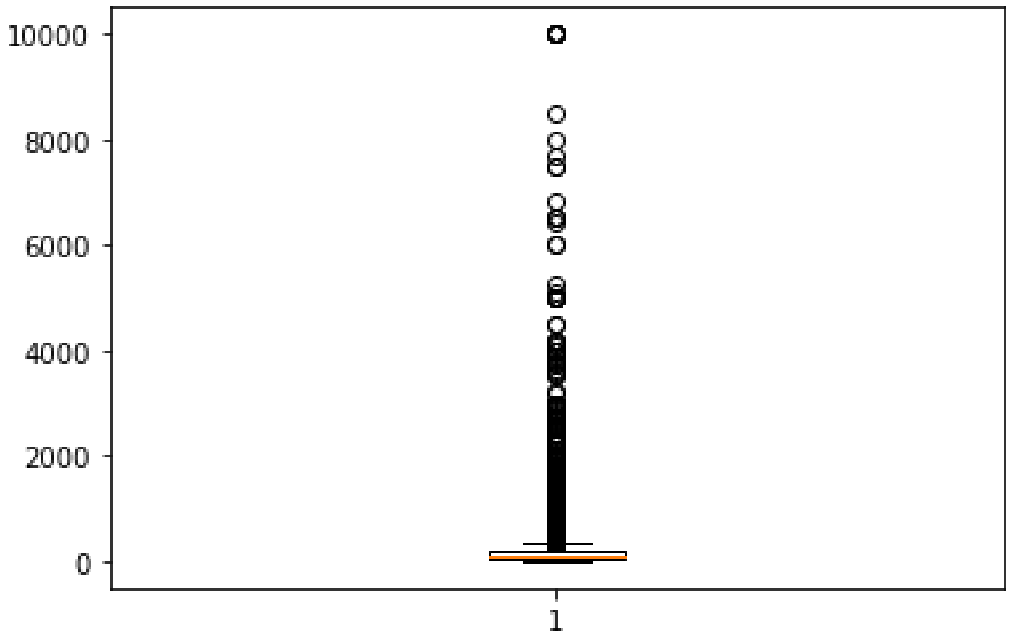
3.0.2 [5 pts] Boxplot 3 features of your choice

- plot boxplots for 3 features of your choice

[114]:
```
# columns = [airbnb["price"], airbnb["minimum_nights"],␣
↪airbnb["availability_365"]]
fig, ax = plt.subplots()
ax.boxplot(airbnb["price"])
```

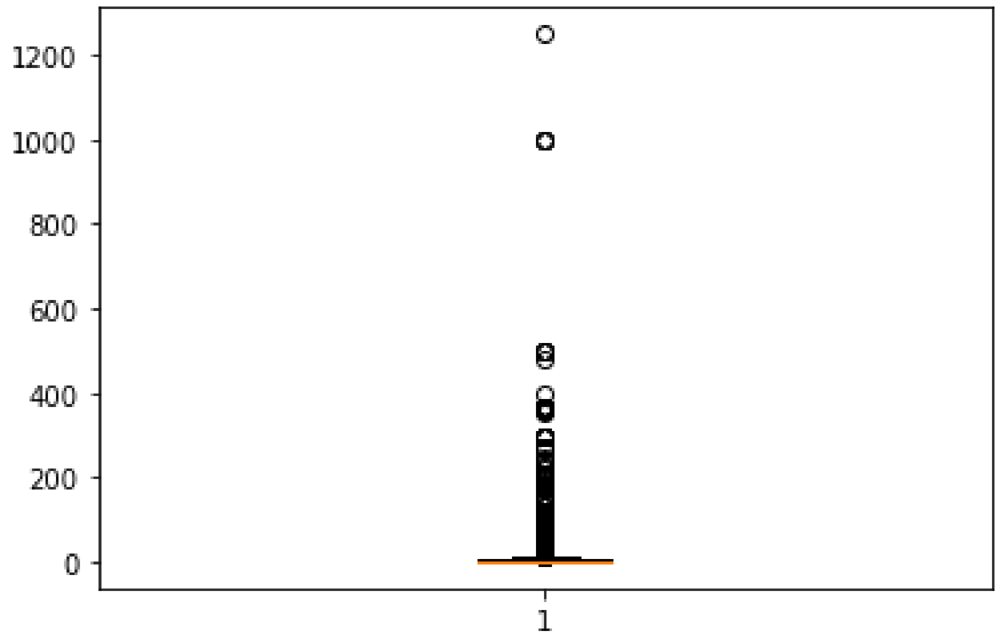
[114]:
```
{'whiskers': [<matplotlib.lines.Line2D at 0x28d76d84700>,
  <matplotlib.lines.Line2D at 0x28d76d84460>],
 'caps': [<matplotlib.lines.Line2D at 0x28d72bbbfd0>,
  <matplotlib.lines.Line2D at 0x28d72bbb340>],
 'boxes': [<matplotlib.lines.Line2D at 0x28d76d84e20>],
```

```
 'medians': [<matplotlib.lines.Line2D at 0x28d72bbb760>],
 'fliers': [<matplotlib.lines.Line2D at 0x28d72bbb160>],
 'means': []}
```



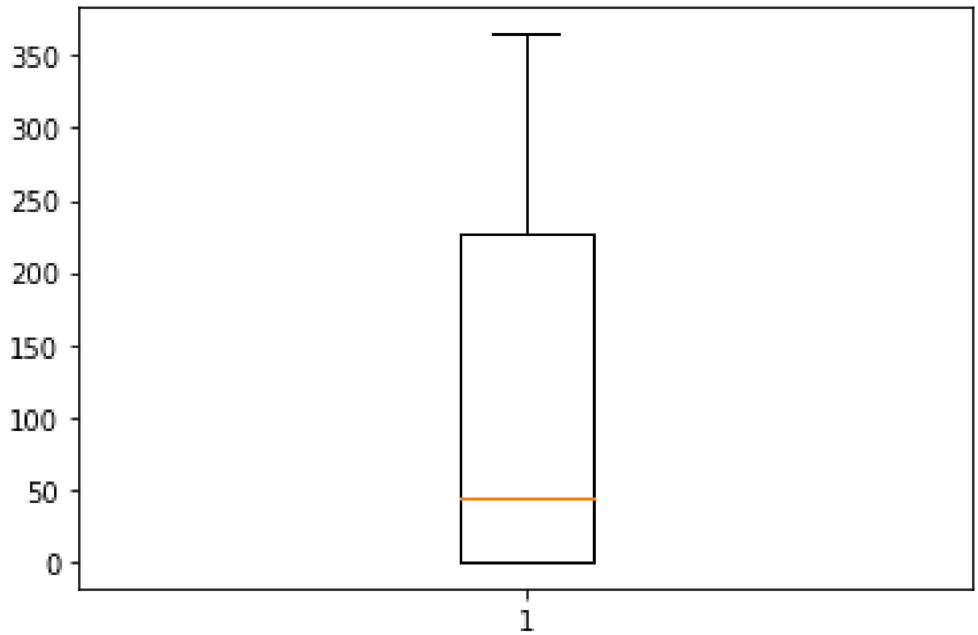
[115]:
```
fig, ax = plt.subplots()
ax.boxplot(airbnb["minimum_nights"])
```

[115]:
```
{'whiskers': [<matplotlib.lines.Line2D at 0x28d727cd250>,
  <matplotlib.lines.Line2D at 0x28d727cd6a0>],
 'caps': [<matplotlib.lines.Line2D at 0x28d727cd7f0>,
  <matplotlib.lines.Line2D at 0x28d727cd790>],
 'boxes': [<matplotlib.lines.Line2D at 0x28d72b42a30>],
 'medians': [<matplotlib.lines.Line2D at 0x28d00005040>],
 'fliers': [<matplotlib.lines.Line2D at 0x28d00005910>],
 'means': []}
```



[116]:
```
fig, ax = plt.subplots()
ax.boxplot(airbnb["availability_365"])
```

[116]:
```
{'whiskers': [<matplotlib.lines.Line2D at 0x28d7280e520>,
  <matplotlib.lines.Line2D at 0x28d7280e130>],
 'caps': [<matplotlib.lines.Line2D at 0x28d027aaaf0>,
  <matplotlib.lines.Line2D at 0x28d027aa3d0>],
 'boxes': [<matplotlib.lines.Line2D at 0x28d7280ec10>],
 'medians': [<matplotlib.lines.Line2D at 0x28d027aa5b0>],
 'fliers': [<matplotlib.lines.Line2D at 0x28d027aa280>],
 'means': []}
```



- describe what you expected to see with these features and what you actually observed

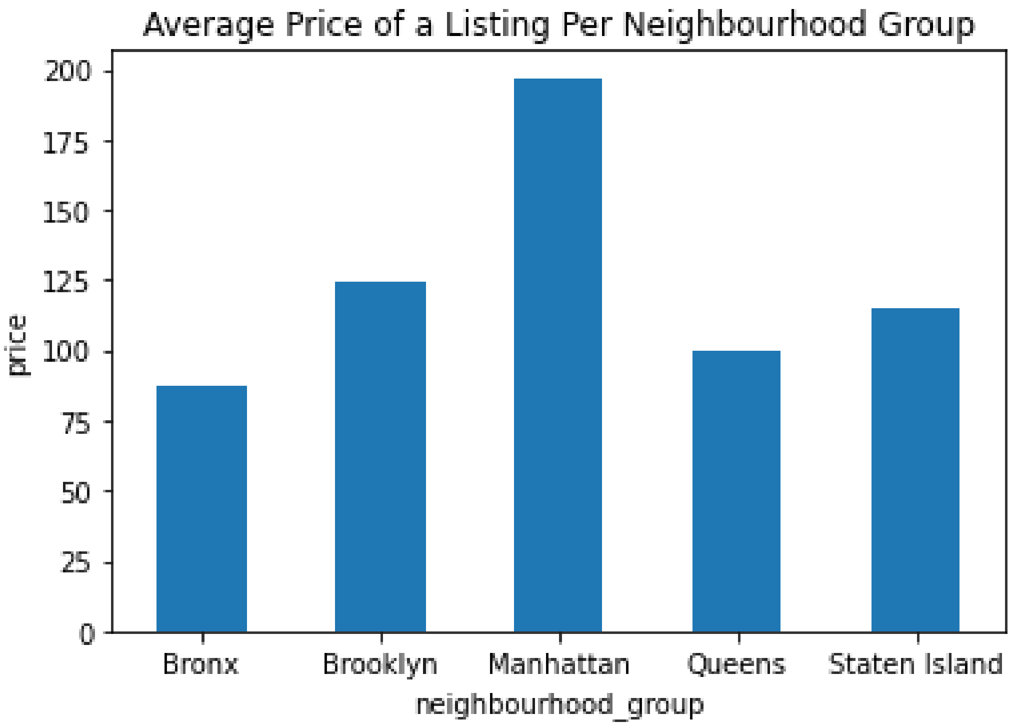
[Response here] * Price: I expected the prices to be skewed to the right since there are a lot of mansions that might cost a lot of money. * Minimum Nights: The number of minimum nights was also skewed to the right. * Availability 365: This field means the number of days a year the Airbnb is available for, and I expected that this value be less than or equal to 365 since there are only 365 days a year.

High variability in price with long tail values, review numbers much more compact, however availability has a wider variance.

3.0.3 [10 pts] Plot average price of a listing per neighbourhood_group

[117]:
```
airbnb_gb_neighbourhood_group = airbnb.groupby(["neighbourhood_group"])
avg_by_neighbourhood_group = airbnb_gb_neighbourhood_group["price"].agg([np.
 ↪average]).reset_index()
avg_by_neighbourhood_group.plot(kind="bar", x="neighbourhood_group",␣
 ↪y="average", ylabel="price", rot=0, legend=False, title="Average Price of a␣
 ↪Listing Per Neighbourhood Group")
```

[117]:
```
<AxesSubplot:title={'center':'Average Price of a Listing Per Neighbourhood
Group'}, xlabel='neighbourhood_group', ylabel='price'>
```



- describe what you expected to see with these features and what you actually observed

[Response here] As we can see from the bar chart above, Manhattan has the most expensive listings on average while the Bronx, in which 30.7% of the population lives below the poverty line, has the cheapest listings on average.

- So we can see different neighborhoods have dramatically different pricepoints, but how does the price breakdown by range. To see let's do a histogram of price by neighborhood to get a better sense of the distribution.

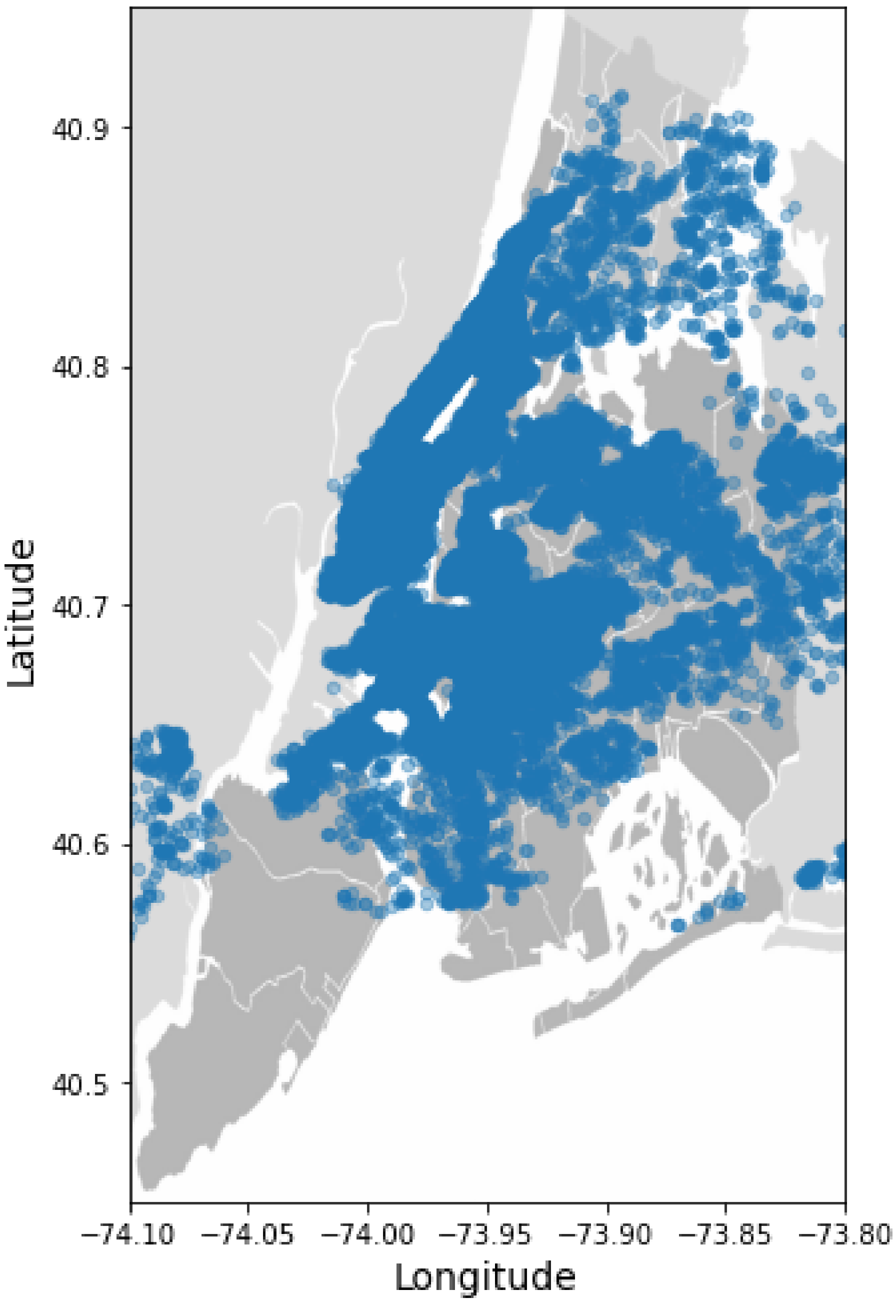
[99]:
```
# see above
```

3.0.4 [5 pts] Plot map of airbnbs throughout New York (if it gets too crowded take a subset of the data, and try to make it look nice if you can :)).

[118]:
```
images_path = os.path.join('./', "images")
os.makedirs(images_path, exist_ok=True)
filename = "newyork.png"

import matplotlib.image as mpimg
newyork_img=mpimg.imread(os.path.join(images_path, filename))
ax = airbnb.plot(kind="scatter", x="longitude", y="latitude", figsize=(10,7),␣
 ↪alpha=0.4)
```

```python
# overlay the New York map on the plotted scatter plot
# note: plt.imshow still refers to the most recent figure
# that hasn't been plotted yet.
plt.imshow(newyork_img, extent=[-74.10, -73.80, 40.45, 40.95], alpha=0.5)
plt.ylabel("Latitude", fontsize=14)
plt.xlabel("Longitude", fontsize=14)

save_fig("newyork_airbnb_plot")
plt.show()
```

```
Saving figure newyork_airbnb_plot
```

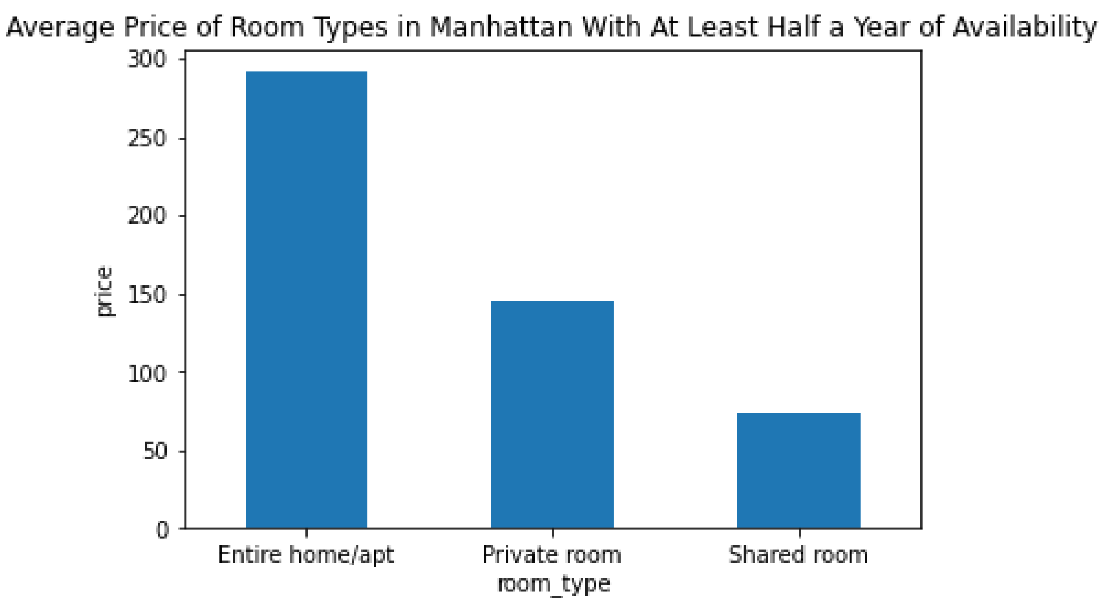


3.0.5 [5 pts] Plot average price of room types who have availability greater than 180 days and neighbourhood_group is Manhattan

[119]:
```
# bar chart
# x-axis room types
# y-axis average price
```

```
filtered_airbnb = airbnb[(airbnb["availability_365"] > 180) &␣
↪(airbnb["neighbourhood_group"] == "Manhattan")]
filtered_airbnb_gb_room_type = filtered_airbnb.groupby(["room_type"])
avg_by_room_type = filtered_airbnb_gb_room_type["price"].agg([np.average]).
↪reset_index()
avg_by_room_type.plot(kind="bar", x="room_type", y="average", ylabel="price",␣
↪rot=0, legend=False, title="Average Price of Room Types in Manhattan With At␣
↪Least Half a Year of Availability")
```

[119]: <AxesSubplot:title={'center':'Average Price of Room Types in Manhattan With At Least Half a Year of Availability'}, xlabel='room_type', ylabel='price'>

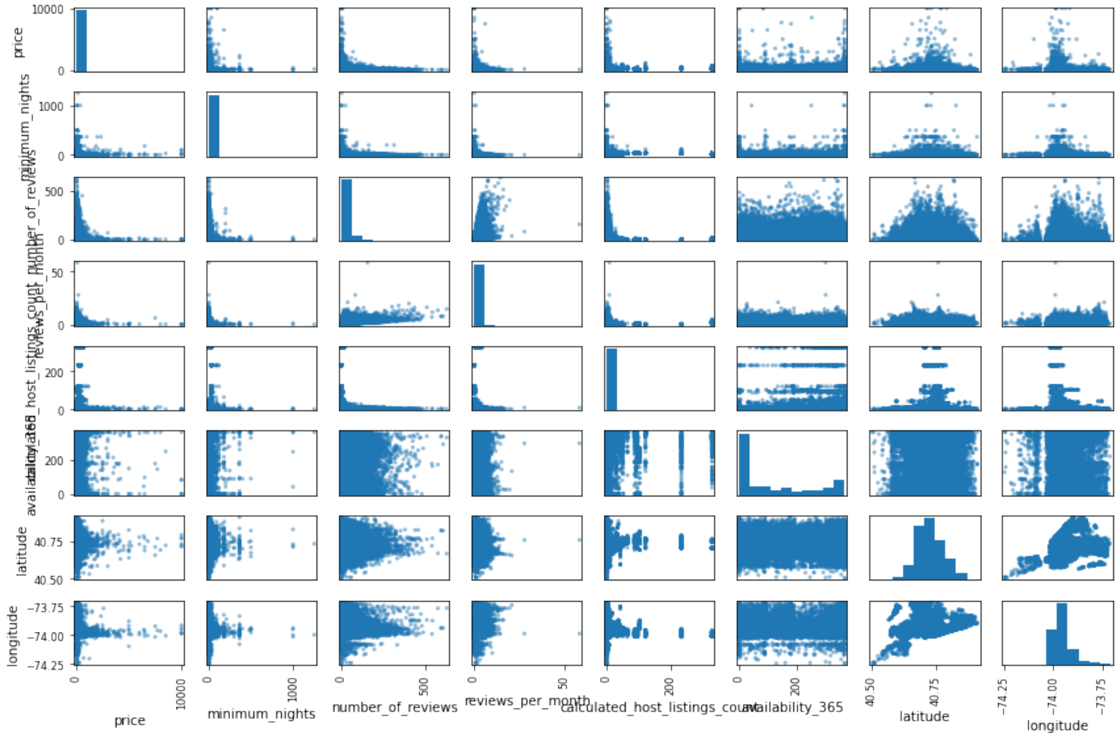


3.0.6 [5 pts] Plot correlation matrix

- which features have positive correlation?
- which features have negative correlation?

[120]:
```
attributes = ["price", "minimum_nights", "number_of_reviews",␣
↪"reviews_per_month", "calculated_host_listings_count", "availability_365",␣
↪"latitude", "longitude"]
#airbnb[attributes].corr()
scatter_matrix(airbnb[attributes], figsize=(12, 8))
save_fig("airbnb_scatter_matrix_plot")
```

Saving figure airbnb_scatter_matrix_plot



[Response here] * number_of_reviews and reviews_per_month have a positive correlation as expected since they are related to one another mathematically. * It also seems like calculated_host_listings_count and availability_365 are positively correlated. A possible explanation for this is that if an airbnb is available more often in the year, the host might have to post each time they need to find a new customer. * availability_365 and reviews_per_month are also positively correlated since more availability means possibility for more customers. This same idea applies for availability_365 and number_of_reviews. * availability_365 and minimum_nights are positively correlated as well. A possible explanation for this is that if an Airbnb is available for larger portion of the year, the host will probably want to set the minimum_nights requirements to be higher so that they don't have to keep changing tenants all the time. * There is a negative correlation between longitude and price, implying that in general, airbnb's to the west (smaller longitude) are more expensive. * There are also more reviews_per_month for airbnb's to the east (greater longitude).

4 [30 pts] Prepare the Data

4.0.1 [5 pts] Augment the dataframe with two other features which you think would be useful

[121]:
```
# There is a positive correlation between availability_per_listing and price.
# If availability_365 is small and the number of listings is large, then this ratio will be small.
```

```python
# The price of the Airbnb will likely be small as well because if it's not␣
↪available for much of the year,
# and the host has to post multiple times, it probably means that there is less␣
↪interest in the Airbnb.
# It makes sense that the low demand might drive the prices down.
airbnb["availability_per_listing"] = airbnb["availability_365"]/
↪airbnb["calculated_host_listings_count"]

# There is a slight positive correlation between minimum_nights_per_listing and␣
↪price.
# A possible explanation for this is that if an Airbnb has a small number of␣
↪minimum nights, and the host
# has listed it multiple times, that probably means the Airbnb is struggling to␣
↪find tenants. Thus, the price
# will likely have to be dropped.
airbnb["minimum_nights_per_listing"] = airbnb["minimum_nights"]/
↪airbnb["calculated_host_listings_count"]

"""
airbnb["rpm_per_longitude"] = airbnb["reviews_per_month"]/airbnb["longitude"]

airbnb["lat_over_long"] = airbnb["latitude"]/airbnb["longitude"]
airbnb["lat_over_rpm"] = airbnb['latitude']/airbnb["reviews_per_month"]
airbnb["long_over_availability"] = airbnb["longitude"]/
↪airbnb["availability_365"]
airbnb["lat_over_availability"] = airbnb["latitude"]/airbnb["availability_365"]
airbnb["rpm_per_listing"] = airbnb["reviews_per_month"]/
↪airbnb["calculated_host_listings_count"]
airbnb["rpm_per_min_night"] = airbnb["reviews_per_month"]/
↪airbnb["minimum_nights_per_listing"]
airbnb["num_reviews_over_rpm"] = airbnb["number_of_reviews"]/
↪airbnb["reviews_per_month"]
airbnb["availability_per_minimum_night"] = airbnb["availability_365"]/
↪airbnb["minimum_nights"]

airbnb["longitude_over_listings"] = airbnb["longitude"]/
↪airbnb["calculated_host_listings_count"]
airbnb["longitude_over_minimum_nights"] = airbnb["longitude"]/
↪airbnb["minimum_nights"]
airbnb["latitude_over_minimum_nights"] = airbnb["latitude"]/
↪airbnb["minimum_nights"]

airbnb["reviews_per_listing"] = airbnb["number_of_reviews"]/
↪airbnb["calculated_host_listings_count"]
airbnb["reviews_over_availability"] = airbnb["number_of_reviews"]/
↪airbnb["availability_365"]
```

```
airbnb["lat_over_listings"] = airbnb["latitude"]/
 ↪airbnb["calculated_host_listings_count"]
"""
# obtain new correlations
corr_matrix = airbnb.corr()
corr_matrix["price"].sort_values(ascending=False)
```

[121]:

```
price                             1.000000
availability_per_listing          0.089530
availability_365                  0.081829
calculated_host_listings_count    0.057472
minimum_nights                    0.042799
minimum_nights_per_listing        0.040203
latitude                          0.033939
id                                0.010619
reviews_per_month                -0.030608
number_of_reviews                -0.047954
longitude                        -0.150019
Name: price, dtype: float64
```

4.0.2 [5 pts] Impute any missing feature with a method of your choice, and briefly discuss why you chose this imputation method

[ ]:

```

```

[134]:

```
sample_incomplete_rows = airbnb[airbnb.isnull().any(axis=1)].head()

# I chose to remove any rows with a null value for "reviews_per_month" because␣
 ↪that likely meant that number_of_reviews was 0,
# and it wouldn't make sense to fill in reviews_per_month with the median value␣
 ↪when the total number of reviews that Airbnb actually
# got was 0.
sample_incomplete_rows.dropna(subset=["reviews_per_month"], inplace=True)
sample_incomplete_rows.head()






airbnb_features = airbnb.drop("price", axis=1) # drop labels for training set␣
 ↪features
                                                # the input to the model␣
 ↪should not contain the true label
airbnb_labels = airbnb["price"].copy()
```

```
# I chose to remove any rows with a null value for "reviews_per_month" because␣
↪that likely meant that number_of_reviews was 0,
# and it wouldn't make sense to fill in reviews_per_month with the median value␣
↪when the total number of reviews that Airbnb actually
# got was 0.
airbnb.dropna(subset=["reviews_per_month"], inplace=True)
airbnb = airbnb[airbnb['calculated_host_listings_count'] != 0]

imputer = SimpleImputer(strategy="median") # use median imputation for missing␣
↪values
categorical = ["neighbourhood", "neighbourhood_group", "room_type"]
airbnb_num = airbnb_features.drop(categorical, axis=1) # remove the categorical␣
↪feature
airbnb_num.head()
```

[134]:

```
     id  latitude  longitude  minimum_nights  number_of_reviews  \
0  2539  40.64749  -73.97237               1                  9
1  2595  40.75362  -73.98377               1                 45
3  3831  40.68514  -73.95976               1                270
4  5022  40.79851  -73.94399              10                  9
5  5099  40.74767  -73.97500               3                 74

   reviews_per_month  calculated_host_listings_count  availability_365  \
0               0.21                               6               365
1               0.38                               2               355
3               4.64                               1               194
4               0.10                               1                 0
5               0.59                               1               129

   availability_per_listing  minimum_nights_per_listing
0                 60.833333                    0.166667
1                177.500000                    0.500000
3                194.000000                    1.000000
4                  0.000000                   10.000000
5                129.000000                    3.000000
```

4.0.3 [15 pts] Code complete data pipeline using sklearn mixins

[141]:

```
# column index
lat_idx, long_idx, min_nights_idx, num_reviews_idx, reviews_per_month_idx,␣
↪listings_idx, availability_idx = 1, 2, 3, 4, 5, 6, 7

#
class AugmentFeatures(BaseEstimator, TransformerMixin):
    '''
    implements the previous features we had defined
```

```
    housing["rooms_per_household"] = housing["total_rooms"]/
↪housing["households"]
    housing["bedrooms_per_room"] = housing["total_bedrooms"]/
↪housing["total_rooms"]
    housing["population_per_household"]=housing["population"]/
↪housing["households"]
    '''
    def __init__(self, add_min_nights_per_listing = True):
        self.add_min_nights_per_listing = add_min_nights_per_listing

    def fit(self, X, y=None):
        return self  # nothing else to do

    def transform(self, X):
        availability_per_listing = X[:, availability_idx] / X[:, listings_idx]
        if self.add_min_nights_per_listing:
            min_nights_per_listing = X[:, min_nights_idx] / X[:, listings_idx]
            return np.c_[X, availability_per_listing, min_nights_per_listing]
        else:
            return np.c_[X, availability_per_listing]

#attr_adder = AugmentFeatures()
#airbnb_extra_attribs = attr_adder.transform(airbnb.values) # generate new␣
↪features

# this will be are numirical pipeline
# 1. impute, 2. augment the feature set 3. normalize using StandardScaler()
num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', AugmentFeatures()),
        ('std_scaler', StandardScaler()),
    ])

#housing_num_tr = num_pipeline.fit_transform(housing_num)

#numerical_features = [x for x in airbnb_num.columns.tolist() if x not in␣
↪categorical]
numerical_features = airbnb_num.columns.tolist()
categorical_features = categorical

full_pipeline = ColumnTransformer([
        ("num", num_pipeline, numerical_features),
        ("cat", OneHotEncoder(), categorical_features),
    ])

airbnb_prepared = full_pipeline.fit_transform(airbnb)
airbnb_prepared
```

[141]: <38843x238 sparse matrix of type '<class 'numpy.float64'>'
	with 582645 stored elements in Compressed Sparse Row format>

4.0.4 [5 pts] Set aside 20% of the data as test test (80% train, 20% test).

[142]:
```
data_target = airbnb['price']
train, test, target, test_target = train_test_split(airbnb_prepared,␣
↪data_target, test_size=0.2, random_state=0)
```

5 [15 pts] Fit a model of your choice

The task is to predict the price, you could refer to the housing example on how to train and evaluate your model using MSE. Provide both test and train set MSE values.

[143]:
```
lin_reg = LinearRegression()
lin_reg.fit(train, target)

# let's try the full preprocessing pipeline on a few training instances
data = test
labels = target_test

print("Predictions:", lin_reg.predict(data)[:5])
print("Actual labels:", list(labels)[:5])
```

```
Predictions: [216.63193311 291.71321556  86.98501491  67.89549236 153.10099963]
Actual labels: [136900.0, 241300.0, 200700.0, 72500.0, 460000.0]
```

[145]:
```
preds = lin_reg.predict(test)
mse = mean_squared_error(test_target, preds)
rmse = np.sqrt(mse)
rmse
```

[145]: 163.3978324537358